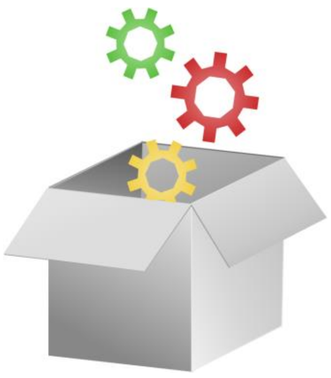
Method SCRIPT SDK Example



Last revision: March ? 2019

Commented [P1]: Tbu

www.palmsens.com

1.1 Contents:

The example “MethodScriptExample.c” found in the “/MethodScriptExample-C” folder demonstrates basic communication with the EmStat Pico through Arduino MKR ZERO using the embedded SDK (C libraries). The example allows the user to start measurements on the EmStat Pico from the PC using a simple C program which makes use of the Method SCRIPT SDK (C libraries).

1.2 Examples:

1.2.1 Example 1: Basic Console Example (MethodScriptExample.c)

This example demonstrates the easiest and most minimalistic way of implementing serial communication to

- Write the input parameters for a measurement (LSV technique) read from a script file
- Read the measurement response from the device.
- Parse the response and display the results on the console.

This does not include error handling, method validation etc.

1.3 Communications

The MSComm.c from the Method SCRIPT SDK (C libraries) acts as the communication object to read/write from/to the EmStat Pico. So the SDK found in the /MethodScriptComm has to be included in the C program as below.

```
#include "MethodScriptComm/MSComm.h"
```

The MSComm is then initialized with the read/write functions as in the code below.

```
MSComm msComm;
RetCode code = MSCommInit(&msComm, &WriteToDevice, &ReadFromDevice);
```

The read/write functions in the C program are mapped to the msComm as shown in the code below.

```
msComm->writeCharFunc = writeCharFunc;
msComm->readCharFunc = readCharFunc;
```

The necessary read/write functions are defined in the C program as below.

```
int WriteToDevice(char c)
{
      char writeChar[2] = {c,'\0'};
      if (WriteFile(hCom, writeChar, 1, &dwBytesWritten, NULL))
      {
            return 1;
      }
      return 0;
}

int ReadFromDevice()
{
      char tempChar;                //Temporary character used for reading
      DWORD noBytesRead;
      ReadFile(hCom,                      //Handle of the Serial port
               &tempChar,                 //Temporary character
               sizeof(tempChar),          //Size of TempChar
               &noBytesRead,              //Number of bytes read
               NULL);
      return (int)tempChar;
}
```

1.3.1 Connecting to the device

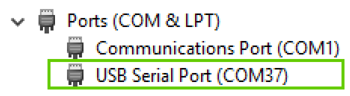
The following example shows how to open a serial com port using the Windows API . Inorder to use the Windows API for serial communication, the windows.h header has to be included in the C program's header file as below.

```
#include <windows.h>
```

The code below can be used to open the com port connected to the devic.

```
HANDLE hCom;
hCom = CreateFile("\\\\.\\COM37", GENERIC_READ | GENERIC_WRITE, 0,  // must
be opened with exclusive-access
              NULL,                     // no security attributes
              OPEN_EXISTING,           // must use OPEN_EXISTING
              0,                       // not overlapped I/O
              NULL              // hTemplate must be NULL for comm devices
              );
```

The name of the com port connected to the device can be found in the Device Manager in Control Panel in Windows as shown below.


Ports (COM & LPT)
Communications Port (COM1)
USB Serial Port (COM37)

Upon successful opening of port, the port connection parameters can be set using the Device control block (DCB) variable as shown in the code below.

```
DCB dcb = { 0 };
// Set up the port connection parameters with the help of device control
block (DCB)
fSuccess = GetCommState(hCom, &dcb);

if (!fSuccess) {
      printf("GetCommState failed with error %lu.\n", GetLastError());
      return 0;
}
// Fill in DCB: 230400 bps, 8 data bits, no parity, and 1 stop bit.
DWORD baud = 230400;
dcb.BaudRate = baud;                   // set the baud rate 230400 bps
dcb.ByteSize = 8;                            // data size
dcb.Parity = NOPARITY;                 // no parity bit
dcb.StopBits = ONESTOPBIT;             // one stop bit

fSuccess = SetCommState(hCom, &dcb);
```

The read/write timeouts are to be set as well as shown below.

```
COMMTIMEOUTS timeouts = { 0 };
timeouts.ReadIntervalTimeout = 1000;            // in milliseconds
timeouts.ReadTotalTimeoutConstant = 1000;       // in milliseconds
timeouts.WriteTotalTimeoutConstant = 50;        // in milliseconds
timeouts.WriteTotalTimeoutMultiplier = 10;      // in milliseconds

if (!SetCommTimeouts(hCom, &timeouts)) {
      printf("SetCommState failed with error %lu.\n", GetLastError());
}
```

1.3.2 Verifying the connected device

Inorder to verify if the device connected is EmStat Pico, the command to get version string “t\n” is sent to the serial port using the WriteStr() in the Method Script SDK (MSComm.c)

```
const char* CMD_VERSION_STRING = "t\n";

//Send the command to get version string and verify if port is connected to
Emstat pico.
char versionString[30];
RetCode code;
BOOL isConnected = 0;
WriteStr(&msComm, CMD_VERSION_STRING);

void WriteStr(MSComm* msComm, const char* buf)
{
      while(*buf != 0)
      {
            WriteChar(msComm, *buf);
            buf++;
      }
}
```

The WriteChar function uses the writeCharFunc() mapped in the msComm to write the input character on the device as shown in the code below.

```
msComm->writeCharFunc(c);
```

The above code in turn redirects to the WritetoDevice() in the example C program. This function writes to the serial port using the Writefile() from the Windows API using the handle created while opening the serial port/device.

```
int WriteToDevice(char c)
{
      char writeChar[2] = {c,'\0'};
      if (WriteFile(hCom, writeChar, 1, &dwBytesWritten, NULL))
      {
            return 1;
      }
      return 0;
}
```

The response from the device is is read character by character until the new line character is read to form the version string. The ReadBuf() from the MSComm library can be used to obtain a line of response from the device. If the version string contains the string “esp” it is identified as EmStat Pico.

```
do{
    code = ReadBuf(&msComm, versionString);
    if(strstr(versionString, "*\n") != NULL &&
        strstr(versionString, "esp") != NULL)
            isConnected = 1;
    if(code == CODE_RESPONSE_END)
            break;
}while(1);
```

The ReadBuf() in the MSComm library inturn uses the read function mapped through the msComm instance to read from the device.

```
tempChar = msComm->readCharFunc();
```

The above code in turn redirects to the ReadFromDevice() in the example C program. It uses the ReadFile() from the Windows API to read a character from the device using the handle created while opening the serial port/device.

```
int ReadFromDevice()
{
      char tempChar;                //Temporary character used for reading
      DWORD noBytesRead;
      ReadFile(hCom,                //Handle of the Serial port
               &tempChar,           //Temporary character
               sizeof(tempChar),    //Size of TempChar
               &noBytesRead,        //Number of bytes read
               NULL);
      return (int)tempChar;
}
```

1.3.3 Sending the method script

The measurement configuration parameters can be read from a script file stored in the PC. In this example, the script file is stored in the same directory as the application executable. The path to the current working directory can be obtained by using the getcwd() as shown below.

The unistd.h library has to be included to make use of getcwd().

```
#include <unistd.h>
```

The macro PATH_MAX from the limits.h library is used to specify the maximum length of the path.

```
char buff[PATH_MAX];
char *currentDirectory = getcwd(buff, PATH_MAX);
char* filePath = "\\ScriptFiles\\ LSV_test_script.txt ";
char* combinedFilePath = strcat(currentDirectory, filePath);
```

The code for opening and reading from the measurement configuration parameters from the script file and writing it on the device are as shown below.

```
FILE *fp;
char str[100];
fp = fopen(fileName, "r");
if (fp == NULL) {
      printf("Could not open file %s", fileName);
      return 1;
}
while (fgets(str, 100, fp) != NULL)       //Reads a single line from the
script file and writes it on the device.
{
      WriteStr(&msComm, str);
}
fclose(fp);
```

1.3.4 Receiving response

The code to receive and parse the response from the device goes on continuously until the measurement response ends. Inorder to read and parse the response from the device, the Receive Package function from the MSComm library can be used.

```
int continueParsing;
do
{
      code = ReceivePackage(&msComm, &data);
      continueParsing = DisplayResults(code);
}while(continueParsing == 1);
```

The ReceivePackage() function in the C library (MSComm.c) reads character by character until end of line ('\n') from the device using the msComm read wrapper - readCharFunc(). The characters are then combined in to a line of response. For example, in this example, every character read (tempChar) is combined to form a line of response in the response buffer string (buf).

```
int tempChar;                    //Temporary character used for reading
int i = 0;
do {
      tempChar = msComm->readCharFunc();
      if(tempChar > 0)
      {
            buf[i++] = tempChar;            // Store tempchar into buffer
            if(buf[0] == (int)'e')
                  return CODE_RESPONSE_BEGIN;
            if(tempChar == '\n')
            {
                  buf[i] = '\0';
                  if(buf[0] == REPLY_MEASURING)
                        return CODE_MEASURING;
                  else if(strcmp(buf, "*\n") == 0)
                        return CODE_MEASUREMENT_DONE;
                  else if(strcmp(buf, "\n") == 0)
                        return CODE_RESPONSE_END;
                  else if(buf[0] == REPLY_MEASURE_DP)
                        return CODE_OK;
                  else
                        return CODE_NOT_IMPLEMENTED;
            }
      }
} while (i < 99);
buf[i] = '\0';
return CODE_NULL;
```

1.3.5 Parsing the response

Each line of response returned by the function **`ReadBuf`**`() in MSComm library`, can be further parsed if it is identified to be a data package. Here's a sample response (raw data) from a Linear sweep voltammetric measurement.

```
eM0000\n
Pda7F85F3Fu;ba48D503Dp,10,288\n
Pda7F9234Bu;ba4E2C324p,10,288\n
Pda806EC24u;baAE16C6Dp,10,288\n
Pda807B031u;baB360495p,10,288\n
*\n
\n
```

While parsing the response, various identifiers are used to identify the type of response packages. For example, In the above sample response package,

1. 'e' marks the beginning of a response.
2. 'M' marks the beginning of a measurement loop.
3. 'P' marks the beginning of a row of data package.
4. "*\n" marks the end of measurement.
5. "\n" marks the end of response.

The following information can be found in the data packages received from the device.

- Potential (set cell potential in V)
- Current (measured current in A)

In case of Impedance spectroscopy measurements, the following data values can be obtained from the response.

- Frequency (set frequency in Hz)
- Real part of complex Impedance (measured impedance Ohm)
- Imaginary part of complex Impedance (measured impedance in Ohm)

The following meta data values if present can also be obtained from the data packages.

- CurrentStatus (OK, underload, overload, overload warning)
- CurrentRange (the current range in use at the moment)
- Noise (Noise)

1.3.5.1 Parsing the parameter values

Each row of data package begins with the header ‘P’. So the pointer is placed next to ‘P’ from where the parsing begins.

```
//Identifies the beginning of the response package
char *P = strchr(responsePackageLine, 'P');
char *packageLine = P+1;

The characters used as delimiters are defined in a char array as below.

const char delimiters[] = " ;\n";
```

The parameters from the data package line can be then split in to tokens based on the delimiters. In this example, the strtokenize function is used repeatedly to separate the parameters from the package line. It splits the line based on the delimiters, returns the start of the split token and holds the pointer to the next token in the array ‘running’. This is being done repeatedly until the function returns null when there are no more tokens or delimiters found or if it is end of the line. The char array ‘running’ is initialized with the response package line received from ReadBuf().

```
char* running = packageLine;  // Initial index of the line to be tokenized
//Pulls out the parameters separated by the delimiters
char* param = strtokenize(&running, delimiters)
do
{
      //Parses the parameters further to get the meta data values if any
      ParseParam(param, retData);
}while ((param = strtokenize(&running, delimiters)) != NULL);//Continues
parsing the response line until end of line

char* strtokenize(char** stringp, const char* delim)
{
  char* start = *stringp;
  char* p;
  //Breaks the string when a delimiter is found and returns a pointer with
starting index, Returns NULL if no delimiter is found
  p = (start != NULL) ? strpbrk(start, delim) : NULL;
  if (p == NULL)
  {
    //The pointer to the successive token is set to NULL if no further
tokens or end of string
```

```
    *stringp = NULL;
  }
  else
  {
    *p = '\0';
    //Saves the pointer to the beginning of successive token to be further
tokenized
    *stringp = p + 1;

  }
  //Returns the current token found
  return start;
}
```

Each of the parameters separated from the package line can be then parsed to get the actual values of the parameters.

The initial two characters of every parameter identifies the parameter.

```
char paramIdentifier[3];
strncpy(paramIdentifier, param, 2); //Splits the parameter identifier string
paramIdentifier[2] = '\0';
```

For example, in the sample package seen above, the parameter identifiers are
'da7F85F3Fu' - 'da' Potential reading and
'ba48D503Dp,10,288' – 'ba' current reading.

The parameter values hold the next 8 characters.

```
strncpy(paramValue, param+ 2, 8);    //Splits the parameter value string
paramValue[9]= '\0';
```

The parameter value for current reading (8 characters)from the above sample package is '48D503Dp'. This value is further parsed to retrieve the actual parameter value with the respective unit prefix.

```
//Retrieves the actual parameter value
parameterValue = (float)GetParameterValue(paramValue);
```

The SI unit prefix from the package can be obtained from the parameter value at position 8

```
//Identify the SI unit prefix from the package at position 8
char charUnitPrefix = paramValue[7];
```

In the above sample package, the unit prefix for current data is 'p' which is 1e-12 A.

The code below parses the actual parameter value excluding the unit prefix (7 characters) and appends the respective prefixes.

```
char strValue[8];
strncpy(strValue, paramValue, 7);
strValue[7] = '\0';
```

The value is first converted from hex to int

```
char *ptr;
int value = strtol(strValue, &ptr , 16);
```

Then value is then adjusted with the Offset value to receive only positive values.

```
const int OFFSET_VALUE = 0x8000000;
```

```
//Values offset to receive only positive values
float parameterValue = value - OFFSET_VALUE;
```

The value of the parameter is returned after appending the SI unit prefix

```
//Return the value of the parameter after appending the SI unit prefix
return (parameterValue * GetUnitPrefixValue(charUnitPrefix));
```

The SI unit prefixes are as follows.

```
const double GetUnitPrefixValue(char charPrefix)
{
	switch(charPrefix)
	{
		case 'a':
			return  1e-18;
		case 'f':
			return 1e-15;
		case 'p':
			return 1e-12;
		case 'n':
			return 1e-9;
		case 'u':
			return 1e-6;
		case 'm':
			return 1e-3;
		case ' ':
			return 1;
		case 'K':
			return 1e3;
		case 'M':
			return 1e6;
		case 'G':
			return 1e9;
		case 'T':
			return 1e12;
		case 'P':
			return 1e15;
		case 'E':
			return 1e18;
	}
	return 0;
}
```

The parameter values are then stored in their respective 'MeasureData' struct (ret_data) member variables based on the identifiers.

The potential readings are identified by the string "da"
The current readings are identified by the string "ba"

```
if(strcmp(paramIdentifier, "da") == 0)
{
	retDdata->potential = parameterValue;
}
else if (strcmp(paramIdentifier, "ba") == 0)
{
	retData->current = parameterValue;
}
```

In case of Impedance sprctroscopy measurement, the following identifiers are used.
The frequency readings are identified by the string “dc”
The real impedance readings are identified by the string “cc”
The imaginary impedance readings are identified by the string “cd”

```
if(strcmp(paramIdentifier, "dc") == 0)
{
      retData->frequency = parameterValue;
}
else if (strcmp(paramIdentifier, "cc") == 0)
{
      retData _data->realImpedance = parameterValue;
}
else if (strcmp(paramIdentifier, "cd") == 0)
{
      ret retData data->imgImpedance = parameterValue;
}
```

After obtaining the parameter identifier and the parameter values from the package, the meta data values if present can be parsed. Meta data values if present are separated by the demiliter character ‘,’

```
//Rest of the parameter is further parsed to get meta data values
ParseMetaDataValues(param + 10, retData);
```

1.3.5.2 Parsing the meta data values

The meta data values are separated based on the delimiter ‘,’ and each of the values is further parsed to get the actual value.

```
const char delimiters[] = ",\n";
char* running = metaDataParams;
char* metaData = strtokenize(&running, delimiters);
```

The first character of each meta data value metaData[0] identifies the type of meta data.

‘1’ – status
‘2’ – Current range index
‘4’ - Noise

The status is 1 character hex bit mask. It is converted to long int. The status can be obtained as shown in the code below.

```
char *ptr;
long statusBits = strtol(&metaDataStatus[1], &ptr , 16);
```

0 indicates OK

```
char* status;
if ((statusBits & 0x0) == STATUS_OK)
      status = "OK";
```

1 indicates overload

```
if ((statusBits & 0x2) == STATUS_OVERLOAD)
      status = "Overload";
```

4 indicates underload

```
if ((statusBits & 0x4) == STATUS_UNDERLOAD)
      status = "Underload";
```

and 8 indicates overload warning (80% of maximum).

```
if ((statusBits & 0x8) == STATUS_OVERLOAD_WARNING)
      status = "Overload warning";
```

For example, in the above sample, the available meta data values for current data are, 10,288. The first meta data value is 10.

1 – meta data status – 0 indicates OK.

The meta data type current range is 2 characters long hex value. If the first bit high (0x80) it indicates a high speed mode current range.

The code below can be used to get current range bits from the package.

```
char  crBytePackage[3];
char* ptr;
//Fetches the current range bits from the package
strncpy(crBytePackage, metaDataCR+1, 2);
```

```
The hex value is then converted to int to get the current range string as
shown below.
```

```
char* currentRangeStr;
int crByte = strtol(crBytePackage, &ptr, 16);

switch (crByte)
{
      case 0:
            currentRangeStr = "100nA";
            break;
      case 1:
            currentRangeStr = "2uA";
            break;
      case 2:
            currentRangeStr = "4uA";
            break;
      case 3:
            currentRangeStr = "8uA";
            break;
      case 4:
            currentRangeStr = "16uA";
            break;
      case 5:
            currentRangeStr = "32uA";
            break;
      case 6:
            currentRangeStr = "63uA";
            break;
      case 7:
            currentRangeStr = "125uA";
            break;
      case 8:
            currentRangeStr = "250uA";
            break;
      case 9:
            currentRangeStr = "500uA";
            break;
      case 10:
            currentRangeStr = "1mA";
            break;
      case 11:
            currentRangeStr = "15mA";
            break;
      case 128:
            currentRangeStr = "100nA (High speed)";
            break;
      case 129:
            currentRangeStr = "1uA (High speed)";
            break;
      case 130:
            currentRangeStr = "6uA (High speed)";
            break;
      case 131:
            currentRangeStr = "13uA (High speed)";
            break;
      case 132:
            currentRangeStr = "25uA (High speed)";
            break;
      case 133:
            currentRangeStr = "50uA (High speed)";
            break;
      case 134:
            currentRangeStr = "100uA (High speed)";
            break;
      case 135:
            currentRangeStr = "200uA (High speed)";
            break;
      case 136:
            currentRangeStr = "1mA (High speed)";
            break;
      case 137:
            currentRangeStr = "5mA (High speed)";
            break;
}
```

For example, in the above sample, the second meta data available is 288.

2 – indicates the type – current range
88 – indicates the hex value for current range index – 1mA. The first bit 8 implies that it is high speed mode current range.

1.3.6 Displaying data on the console window

The data values and meta data values stored in the ‘MeasureData’ struct object ‘retData’ can be then written on the serial port in the arduino code based on the return code from ReceivePackage(). The return code from ReceivePackage() is based on the identified char/string found in the response package.

‘e’ indicates the beginning of a response and the parsing continues.

```
if(code == CODE_RESPONSE_BEGIN)
{
  //do nothing
}
```

‘M’ it indicates the beginning of a measurement loop response

```
else if(code == CODE_MEASURING)
{
  printf("\nMeasuring... \n");
}
```

'P' indicates the beginning of data package and CODE_OK implies successful parsing of data package.

```
else if(code == CODE_OK)
{
  if(nDataPoints == 0)
  printf("\nReceiving measurement response:\n");
  //Received valid package, print it.
  printf("\n %d \t", ++nDataPoints);
  printf("E(V): %6.3f \t", data.potential);
  fflush(stdout);
  printf("i(A) : %11.3E \t", data.current);
  printf("Status: %-15s  ", data.status);
  printf("CR: %s ", data.cr); }
  fflush(stdout);
}
```

"*\n" indicates the end of a measurement loop.

```
else if(code == CODE_MEASUREMENT_DONE)
{
  printf("\nMeasurement completed. ");
}
```

'\n' indicates the end of response.

```
else if(code == CODE_RESPONSE_END)
{
  printf("%d data point(s) received.", nDataPoints);
  fflush(stdout);
}
```

Error message in case parsing fails.

```
else{
  //Failed to parse or identify package.
  printf("\nFailed to parse package: %d\n", code);
}
```